BREADTH FIRST SEARCH

ISTANZA: G = (V, E) *(orientato o meno)*, s $\in$ V *(vertice sorgente)*

COSA FA

1. Scopre i vertici ***raggiungibili*** dal vertice sorgente *s*.
2. Calcola $\forall$ v $\in$ V la ***distanza*** di v dal vertice sorgente *s*.
3. *(Principio di funzionamento)*
   Amplia la frontiera tra vertici già scoperti e vertici non ancora scoperti in modo tale che:
   - prima scopre i vertici a distanza 0 da *s (ovvero s stesso)*,
   - poi scopre i vertici a distanza 1 da *s*,
   - poi scopre i vertici a distanza 2 da *s* e così via fino a raggiungere la profondità massima del sotto grafo di *s*.
4. Genera un ***albero radicato*** *(Albero BFS)* con radice *s* contenente tutti i vertici raggiungibili da *s (tramite le informazioni contenute in π)*.
   ALBERO BFS
   Denotiamo l'albero BFS con **G$_{\pi}$ = (V$_{\pi}$, E$_{\pi}$)** tale che:
   V$_{\pi}$ contiene tutti i vertici raggiungibili dalla sorgente *s*.
   Possiamo quindi definire V$_{\pi}$ in 3 modi equivalenti, ovvero:
   - V$_{\pi}$ = { v $\in$ V | v.col $\neq$ WHITE } *(oppure v.col = BLACK)*
   - V$_{\pi}$ = { v $\in$ V | v.d $\neq$ ∞ }
   - V$_{\pi}$ = { v $\in$ V | v.π $\neq$ NIL } $\cup$ {*s*}

   E$_\pi$ sono gli archi dell'albero.
   Definiamo E$_{\pi}$ come
   - E$_{\pi}$ = { (v.π, v) | v $\in$ V$_\pi$ - {*s*} }
   - E$_\pi$ = { (u, v) | u, v $\in$ V$_{\pi}$ - {s} : v.π = u} $\cup$ { (*s*, w) | w $\in$ Adj[*s*] } *(non sono sicuro della correttezza di questa)*

ALGORITMO

Uso dei colori per dire se un certo nodo è stato scoperto, completamente esplorato e non scoperto;

- WHITE - Non scoperto.
- GREY - Scoperto ma non completamente esplorato, non ho ancora scoperto tutta la sua lista di adiacenza.
- BLACK - Scoperto e completamente esplorato, ho scoperto la sua lista di adiacenza.

Inoltre salvo un'informazione relativa a chi ha scoperto un certo vertice *v* in un suo parametro π *(parent)*. Ovviamente la sorgente non ha alcun parent.

Se mi serve sapere solo i nodi raggiungibili dalla sorgente, ovviamente non ho bisogno dei colori o dei predecessori. Queste sono informazioni che ci servono per ulteriori implementazioni dell'algoritmo.

BFS(G, s) {
INIZIALIZZAZIONE Tempo $\Theta(|V|)$ visto che inizializziamo tutti i v $\in$ V.
for all *u* $\in$ V - {*s*}
  u.col = WHITE

```
    u.d = $\infty$
    u.$\pi$ = NIL
s.col = GREY
s.d = 0
s.$\pi$ = NIL
enqueue(Q, s)
```

ESECUZIONE Tempo $O(|E|)$ visto che percorro solo gli archi raggiungibili da *s (potrebbero essere tutti)*

```
while Q $\neq \empty$
    u = dequeue(Q)
    for all v $\in$ Adj[u]
        if v.col == WHITE
            v.col = GREY
            v.d = u.d + 1
            v.$\pi$ = u
            enqueue(Q, v)

    u.col = BLACK
}
```

Nella coda spesso ci saranno dei nodi a distanza *k* insieme a dei nodi a distanza *k + 1*, una volta levati quelli a distanza *k* avrò solo quelli a distanza *k + 1* se esistono, altrimenti avrò la coda vuota fermando il ciclo while.

ESERCIZI

CONTARE VERTICI RAGGIUNGIBILI

Dato un grafo G = (V, E) x $\in$ V, contare i vertici raggiungibili da x.

```
BFS(G, x)
n = 0
for all v $\in$ V
   if v.col != WHITE (oppure v.d != $\infty$)
      n++
Return n
```

Ovviamente la BFS esplora solo i nodi raggiungibili dalla sorgente, lasciando bianchi quelli non raggiungibili da essa *(e quindi con la distanza pari ad ∞)*, quindi mi basta contare quanti nodi esplorati ottengo alla fine della BFS.

STABILIRE SE UN GRAFO N OR È UN ALBERO

Dato un grafo non orientato scrivere un algoritmo che decide se il grafo è un albero.
Sappiamo che un albero è un grafo non orientato, connesso e aciclico; ci dicono già che è non orientato, dobbiamo stabilire se è connesso e aciclico.

DECIDERE SE È CONNESSO O NO
Basta estrarre un vertice a caso e lanciare BFS su quel vertice; se alla fine dell'esecuzione ci sono dei vertici bianchi o con distanza = ∞ il grafo sicuramente non è connesso.

```
isConnected(G) {
s = Random(V)
BFS(G, s)
for all v $\in$ V
   if v.d == $\infty$
      return FALSE
return TRUE
}
```

DECIDERE SE È ACICLICO

```
BFS(G, s) {
for all u $\in$ V - {s}
   u.col = WHITE
s.col = GREY
enqueue(Q, s)

while Q $\neq \empty$
   u = dequeue(Q)
   for all v $\in$ Adj[u]
      if v.col == WHITE
         v.col = GREY
         enqueue(Q, v)
      else if v.col == GREY
         Return FALSE
```

Return TRUE
}

DECIDERE SE È UN ALBERO
Per decidere quindi se un grafo è un albero potremmo unire le due informazioni ottenute fino ad ora. Tuttavia, possiamo anche sfruttare un'altra definizione di albero e, dopo aver controllato che il grafo si connesso, confrontare il numero di vertici e di lati.

```
BFS(G, s) {
for all u $\in$ V - {s}
   u.d = $\infty$
s.d = 0
enqueue(Q, s)
e = 0

while Q $\neq \empty$
   u = dequeue(Q)
   for all v $\in$ Adj[u]
      e++ (e contiene dunque il doppio del numero di archi all'interno della CC di cui fa parte s)
      if v.d == $\infty$
         v.d = u.d + 1
         enqueue(Q, v)
}
for all v $\in$ V
   if v.d == $\infty$
      Return FALSE

n = Adj.length (n = |V| !!!)
if e/2 == n - 1
   Return TRUE
else
   Return FALSE
```

ATTENZIONE: *e* alla fine conterrà il doppio del numero degli archi contenuti nella CC di cui fa parte *s* visto che stiamo trattando dei grafi non orientati e quindi all'interno della lista di adiacenza di ogni nodo ci sarà anche il nodo parent.

STABILIRE SE UNA CC È UN ALBERO

Dato un grafo G = (V, E) non orientato e dato un vertice *x* $\in$ V, stabilire se la CC contenente *x* è un albero.

Esercizio più facile di quello di prima, infatti se consideriamo una sola componente connessa, è ovviamente ***connessa***; ci basta confrontare il numero di archi con il numero di vertici per dire se sia un albero o meno.

```
BFS(G, s) {  (s sarebbe il nostro x in questo caso)
for all u $\in$ V - {s}
   u.d = $\infty$
s.d = 0
enqueue(Q, s)
e = 0, nv = 0
```

```
while Q $\neq \empty$
   u = dequeue(Q)
   nv++   (ogni volta che tiro fuori un vertice dalla coda lo conto)
   for all v $\in$ Adj[u]
      e++
      if v.d == $\infty$
         v*.d = u.d + 1
         enqueue(Q, v)

if e/2 == nv - 1
   Return TRUE
else
   Return FALSE
}
```

NODI A DISTANZA $\leq$ K

Dato un grafo G = (V, E), un vertice *s* $\in$ V ed un valore *k* > 0 intero, modificare BFS in modo tale che scopra solo i vertici a distanza $\leq$ *k* da *s*

```
BFS(G, s, k) {
for all u $\in$ V - {s}
   u.d = $\infty$
s.d = 0
enqueue(Q, s)

while Q $\neq \empty$
   u = dequeue(Q)
   for all v $\in$ Adj[u]
      if v.d == $\infty$
         v.d = u.d + 1
         if v.d < k          (non accodo più una volta che ho raggiunto k - 1 livelli)
            enqueue(Q, v)
}
```

ATTENZIONE: Mi fermo a k - 1 poiché una volta accodati i nodi a livello k - 1, andrò a scoprire solo i loro figli, ovvero i nodi a distanza k, per poi avere la coda vuota e terminare il ciclo.

GRAFI COMPLETI

Dato un grafo non orientato G = (V, E), dato *s* $\in$ V, scrivere un algoritmo che stabilisce se la CC contenente *s* è un sotto grafo che, preso singolarmente, è un grafo completo.

Nel caso dei grafi non orientati purché un grafo sia completo deve avere il massimo numero di archi, ovvero $\sum \limits_{i = 1}^{n} n - i$ = $\frac{(n - 1)\space * \space n}{2}$

```
BFS(G, s) {
for all u $\in$ V - {s}
   u.d = $\infty$
s.d = 0
enqueue(Q, s)
e = 0, nv = 0

while Q $\neq \empty$
   u = dequeue(Q)
   nv++
   for all v $\in$ Adj[u]
      e++
      if v.d == $\infty$
         v.d = u.d + 1
         enqueue(Q, v)

if e/2 == ((nv - 1) * nv)/2
   Return TRUE
else
   Return FALSE
}
```

PER GRAFI ORIENTATI

Il procedimento è lo stesso, ma nei grafi orientati il numero di archi deve essere pari a nv^{nv}.